# Compatibility of Structures and Unions Without Tag

Author: Martin Uecker

Charter Principles (n3280):
	Keep the language small and simple
	Do not leave features in an underdeveloped state
	Enable secure programming

## 1. Background

In C23, WG14 adopted new tag compatibility rules that make it possible redeclare structures, union, and enumerations and use them together when they have same tag. These rules simplify the common approach of using macros to implement type-safe generic data types in C (see the Appendix for an example).

These rules did not enter ISO C23 exactly as proposed. During standardization, the committee observed that code such as the following would be became valid, reducing type safety.

```
typedef struct { double value; } celsius_t;
typedef struct { double value; } fahrenheit_t;

fahrenheit_t x = { 451. };
celsius_t y = x;
```

For this reason, the rules were limited to types declared with tag. As discussed in *N3332: _Record types* and revisited in the following, this prevents users from fully exploiting the potential of the new structural compatibility rules. N3332 proposes *record types* as a new language feature to address this problem. Here, based on the observation that the current rules also still leave room for improvements, and it is therefor suggested to further revise the rules along the lines of the original proposal and to address the type safety issue directly in the context of type definitions.

### Issue 1: Usability of Generic / Algebraic Types

When constructing a generic type using a macro, adding a tag as in the following examples causes errors when one tries to use multiple such vector types in the same scope.

```
#define vec(T) struct vec { size_t len; typeof(T) data[/* .len */]; }
```

To fix this problem, one has to synthesize a unique tag.

```
#define vec(T) struct vec_##T { size_t len; typeof(T) data[/* .len */]; }

struct item;
vec(struct item *) *v;
```

This solution is not ideal when the type argument consists of several tokens, because then the macro expands to nonsense. As a workaround, the user has to create a typedef name for the element type.

```
struct vec_struct item * { .. } *v;     // error
typedef struct item *element_t;
vec(element_t) *v;                      // ok
```

It has been noted by users that making tagless structures compatible would allow a convenient construction of generic types such as pointers or other generic types as element typ without the need for this workaround. In particular, this would then also automatically ensure that different such vectors with the same element type are always compatible, making them suitable as vocabulary types used for interchange between different modules from different vendors, while the present requirement to introduce a new typedef name may introduce accidental incompatibility when different names are chosen. Unfortunately, the tagless version

```
#define vec(T) struct { size_t len; typeof(T) data[/* .len */]; }
```

currently does not work when compatibility of identical vector types is required, as each such structure defines a different and incompatible type.

```
struct item;
void work(vec(struct item *) *);

void test()
{
	vec(struct item *) *p = calloc(1, sizeof *p);
	if (!p) abort();
	work(p);		// argument not compatible!
	free(p);
}
```

As a side note, I like to mention that there exists an obscure alternative to using tagless types in this situation – but only when the statement expression extension is available. Then one can simply hide the tag in a nested scope.

```
#define HIDE(T) typeof(({ (T){ }; }))
#define vec(T) HIDE(struct vec { size_t len; typeof(T) data[/* .len */]; })
```

Unfortunately, this solution also has limitations, as it fails at file scope where statement expression are currently not supported by compilers. A future version of statement expression that work at file-scope could address this problem, but this does not seem to be within reach and in any case would remain an obscure workaround a user needs to be aware of.

**Issue 2: Usability of Nested Structures or Unions**

The current rules cause other usability issues. They imply that having a member without tag also makes a containing structure incompatible and even non-identical (not the same type) to a syntactically identical declaration. This may be surprising to users.

```
struct foo { struct { int value; } x; int y; };
struct foo { struct { int value; } x; int y; };   // not compatible!

struct bar { struct { int value; }; int y; };     // anonymous member
struct bar { struct { int value; }; int y; };
```

GCC supports the second case (which arguably already follows from the rules), while Clang intentionally supports also the first case to improve usability.

**Issue 3: Useless Declarations in Function Prototypes**

Another remaining issue is that tagless structures and unions that are not declared inside typedefs are useless in some situations. In particular, when such a type is declared inside a function prototype, the function can not be called at all because the unique type can not be constructed in the caller.

```
int foo(struct { double value; } x);
```

```
<source>:3:9: warning: anonymous struct declared inside parameter list will
not be visible outside of this definition or declaration
    3 | int foo(struct { double value; } x);
```

**Issue 4: Inconsistency Between Same and Different TU**

Finally, while the current rules make the following two type incompatible inside a single translation unit, they are still compatible across translation units, leaving an annoying inconsistency that complicates whole program analysis and refactoring where code is moved between translation units.

```
typedef struct { double value; } celsius_t;
typedef struct { double value; } fahrenheit_t;
```

## 2. Proposal

It is proposed to make also tagless types fully compatible in the general case as originally proposed, and instead address the type safety issue in a different way. The rationale is that tagless structures and unions that are not declared inside typedefs are otherwise mostly useless. The proposed changes would further simplify the language, make it more consistent, and resolve all the aforementioned issues.

The only case where it is indeed undesirable is the typedef situation, i.e. the situation the committee originally had concerns about. To address this, we take inspiration from *N3320: Strong Typedefs* that explains "*Strongly-typed* typedefs behave as though a unique, virtual qualifier is added onto a type at the position within the type derivation that the typedef name was used." Instead of a virtual qualifier, we propose that the typedefs behave similar as if a virtual tag that is derived from the typedef name in a unique way is added as in the following example.

```
typedef struct __celsius_t { double value; } celsius_t;
typedef struct __fahrenheit_t { double value; } fahrenheit_t;
```

With the proposed change redefinitions of typedef names for tagless structures are then also allowed.

```
typedef struct __celsius_t { double value; } celsius_t;
typedef struct __celsius_t { double value; } celsius_t; // ok
```

When using type-generic macros one sometimes uses local typedef names to avoid expanding a macro argument multiple times, hence this exception should only apply when the structure is explicitly declared inside a typedef declaration and should not apply when a previously declared type is referenced via a typedef name or using typeof.

**Comparison to N3332**

N3332 proposes alternative solution to the first issue by adding a new keyword and syntax to declare compatible types without tag.

```
#define vec(T) struct _Record { size_t len; typeof(T) data[]; }
```

This addresses the main issue (1) in a clean way, but would not resolve the other minor usability and consistency issues (2-4).

In addition to this syntax, N3332 also proposed additional variants, i.e. `_Record(types)` to be able to ignore the names of the members when determining type compatibility, and to support other implementation-defined extensions.

**Future Work**

The rules for "same type" and typedef redeclarations are – in general – still not very clear. In fact, perhaps the notion of "same type" should even be removed, as it is used only for typedef and tag redeclarations (and with the proposed changes it is clear how similar these are) – basing the type system fully on the single concept of type compatibility. It would also be desirable to allow typedef redeclarations in more cases as previously discussed. A nice benefit of this which is already partially realized with the C23 rules is that one can now often omit include guards for headers because all contained declarations are idempotent – removing another annoyance from the language.

## 3. Wording Changes (N3783)

### 3.1 Compatibility of Types Without Tag in the Same TU

6.2.7 Compatible type and composite type

1 Two types are compatible types if they are the same. Additional rules for determining whether two types are compatible are described in 6.7.3 for type specifiers, in 6.7.4 for type qualifiers, and in 6.7.7 for declarators.[45)] Moreover, two complete structure, union, or enumerated types **that are** declared with the same tag **or that both are declared without tags** are compatible if the members satisfy the following requirements:

— there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types;
— if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier;
— and, if one member of the pair is declared with a name, the other is declared with the same name.

For two structures, corresponding members shall be declared in the same order. For two unions declared in the same translation unit, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations corresponding members shall have the same values; if one has a fixed underlying type, then the other shall have a compatible fixed underlying type. For determining type compatibility, anonymous structures and unions are considered a regular member of the containing structure or union type, and the type of an anonymous structure or union is considered compatible with the type of another anonymous structure or union, respectively, if their members fulfill the preceding requirements. Furthermore, two structure, union, or enumerated types declared in separate translation units are compatible in the following cases:

~~**— both are declared without tags and they fulfill the preceding requirements;**~~
— both have the same tag and are completed somewhere in their respective translation units and they fulfill the preceding requirements;
— both have the same tag and at least one of the two types is not completed in its translation unit. Otherwise, the structure, union, or enumerated types are incompatible.[46)]

## 3.2 Type Safety for Type Definitions

6.7.9 Type definitions

Semantics

**3 If a type definition uses a structure, union, or enumeration specifier without tag, then it behaves as if it had been declared with a tag that is uniquely derived from the declared identifier by prefixing it with a double underscore.**[XXX)]

**EXAMPLE 3 The following type definitions specify incompatible structure types.**

```
typedef struct { double value; } celsius_t;
typedef struct { double value; } fahrenheit_t;
```

**EXAMPLE 4 The following type definitions are defined as the same structure type.**

```
typedef struct { double value; } celsius_t;
typedef struct { double value; } celsius_t;
```

**XXX) This affects type compatibility (6.2.7) and redefinition of typedef names (6.7.1) and redeclarations of tags (6.7.3.4) accordingly. An explicit declaration of a tag prefixed with a double underscore in a program has undefined behavior.**

## 3.3 Redeclarations of Types

6.7 Declarations
6.7.1 General

4 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:

— a typedef name can be redefined to denote the same type as it currently does, provided that type is not a variably modified type;
— enumeration constants and tags can be redeclared as specified in 6.7.3.3 and 6.7.3.4, respectively

6.7.3.4 Tags

Constraints

1 Where two declarations ~~**that**~~ **have the same scope and** use the same tag, **they shall** declare the same type. ~~**, they shall both use the same choice of struct, union, or enum. If two declarations of the same type have a member-declaration or enumerator-list, one shall not be nested within the other and both declarations shall fulfill all requirements of compatible types (6.2.7) with**~~

~~the additional requirement that corresponding members of structure or union types shall have the same (and not merely compatible) types.~~

Semantics

4 ~~All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type.~~ **All declarations of structure, union, or enumerated type declare the same type, if they fulfill all requirements of compatible types (6.2.7) with the additional corresponding members of structure or union types have the same (and not merely compatible) types.**

~~9 Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type~~

7 EXAMPLE 1 The following example shows allowed redeclarations of the same structure, union, or enumerated type in the same scope:

```
….
typedef struct { int x; } q_t;
typedef struct { int x; } q_t;          // same type

struct S { int x; };
void foo(void)
{
        struct T { struct S s; };
        struct S { int x; };
        struct T { struct S s; };       // struct S is the same type
}
….
```

8 EXAMPLE 2 The following example shows invalid redeclarations of the same structure, union, or enumerated type in the same scope:

….

~~typedef struct { int x; } q_t;~~
~~typedef struct { int x; } q_t; // not the same type~~

~~struct S { int x; };~~
~~void foo(void)~~
~~{~~
~~struct T { struct S s; };~~
~~struct S { int x; };~~
~~struct T { struct S s; }; // struct S not the same type~~
~~}~~

….

6.7.3.2 Structure and union specifiers

**EXAMPLE 3 The following illustrates a valid redeclaration of a tag for a structure type that contains another nested declaration of a structure without tag.**

```
struct foo { struct { int value; } x; };
struct foo { struct { int value; } x; };          // valid redeclaration
```

## 4. Appendix: Example for a Generic Vector Type

Note that in this toy example the unsigned size computation suffer from silent unsigned wraparound and allocation failures are handled by calling abort(). A more realistic example would only be slightly more complex.

https://godbolt.org/z/f89eGdrTo

```c
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>

#define vec(T) struct vec_##T { size_t len; typeof(T) data[/* .len */]; }

#define vec_eltype(v) typeof((v)->data[0])
#define vec_array(v) (*(vec_eltype(v)(*)[(v)->len]){ &(v)->data })
#define vec_length(v) (_Countof(vec_array(v)))
#define vec_sizeof(v) (sizeof(v) + vec_length(v) * sizeof(vec_eltype(v)))
#define vec_push(v, e)                          \
do {                                            \
      auto _v = (v);                            \
      (*_v)→len++;                              \
      *_v = realloc(*_v, vec_sizeof(*_v));      \
      if (!*_v) abort();                        \
      (*_v)->data[(*_v)->len - 1] = (e);        \
} while(0)


// usage example

int main()
{
      vec(int) *p = calloc(1, sizeof *p);
      if (!p) abort();

      for (int i = 0; i < 10; i++)
            vec_push(&p, i);

      for (size_t i = 0; i < vec_length(p); i++)
            printf("%d\n", vec_array(p)[i]);

      free(p);
}
```